Reingeniería del Paradigma de Software

Un Análisis del Ciclo de Vida de Desarrollo Impulsado por IA (AI-DLC) y la Propuesta Técnica de Kiro IDE

Este documento fue construido usando la funcionalidad DeepResearch de Gemini y curado por el equipo de Hardcore AI.

Introducción: La Crisis del "Vibe Coding" y la Necesidad de Estructura

La industria del desarrollo de software se encuentra en una encrucijada histórica. Desde la popularización de los Grandes Modelos de Lenguaje (LLMs) a principios de la década de 2020, la ingeniería de software ha experimentado una aceleración sin precedentes en la generación de código. Herramientas pioneras como GitHub Copilot y, más tarde, editores como Cursor, introdujeron lo que coloquialmente se ha denominado "Vibe Coding" o "programación por sensaciones". Este paradigma se caracteriza por una interacción fluida, basada en chat, donde el desarrollador solicita fragmentos de lógica o refactorizaciones mediante lenguaje natural, y la Inteligencia Artificial (IA) responde con soluciones inmediatas.

Sin embargo, a medida que la adopción de estas herramientas ha madurado, han comenzado a emerger grietas significativas en este modelo operativo, especialmente cuando se aplica a sistemas empresariales complejos y de larga duración.

La propuesta central de AI-DLC y respaldada por investigaciones y lanzamientos recientes de Amazon Web Services (AWS), argumenta que el enfoque actual de "asistencia por IA" es insuficiente para las demandas de la ingeniería de software moderna. Los autores de la metodología, entre ellos los arquitectos de soluciones principales Raja SP, Siddhesh Jog y Will Matos, postulan que la industria debe transicionar de un modelo donde la IA es un "copiloto" a uno donde la IA es el motor de ejecución principal, gobernado por un marco de trabajo riguroso y estructurado. Este nuevo paradigma ha sido bautizado como AI-Driven Development Lifecycle (AI-DLC) o Ciclo de Vida de Desarrollo Impulsado por IA.

El problema fundamental que el AI-DLC busca resolver es la falta de determinismo, trazabilidad y gobernanza en el "Vibe Coding". Cuando un equipo de desarrollo depende excesivamente de chats efímeros y contextos volátiles, se produce una degradación de la arquitectura del software. La documentación se vuelve obsoleta casi instantáneamente, las decisiones de diseño quedan enterradas en historiales de chat no estructurados y la deuda técnica se acumula a una velocidad proporcional a la generación de código. La propuesta del AI-DLC, materializada técnicamente a través del entorno de desarrollo Kiro IDE, sugiere que la solución no es menos IA, sino una IA más disciplinada, restringida por especificaciones formales ("Specs") y archivos de dirección ("Steering Files") que actúan como contratos inmutables entre la intención humana y la ejecución de la máquina. Este informe desglosa de manera exhaustiva el estado actual de esta propuesta, analizando sus fundamentos teóricos, su implementación técnica en Kiro IDE, la recepción del mercado frente a competidores establecidos como Cursor y Windsurf, y las implicaciones económicas y operativas para el futuro de la ingeniería de software.

1. El Marco Teórico del AI-DLC: Redefiniendo el Ciclo de Vida

El Ciclo de Vida de Desarrollo Impulsado por IA (AI-DLC) no es simplemente una colección de herramientas, sino una metodología formal que busca reemplazar o aumentar significativamente el Ciclo de Vida de Desarrollo de Software (SDLC) tradicional y las metodologías Ágiles. Mientras que el SDLC tradicional (Waterfall, Agile, Scrum) fue diseñado para optimizar la coordinación del esfuerzo humano, el AI-DLC está diseñado para optimizar la supervisión humana sobre la ejecución de la IA.

1.1 Filosofía Central: Ejecución Impulsada por IA con Supervisión Humana

La premisa central del documento analizado es que los humanos no escalan bien en tareas de generación repetitiva, pero son insustituibles en tareas de juicio crítico, contexto de negocio y empatía con el usuario final. Por el contrario, la IA posee una capacidad casi infinita para la generación de sintaxis y patrones, pero carece de contexto y juicio moral o comercial intrínseco.

El AI-DLC propone invertir la pirámide de trabajo tradicional:

1. **Modelo Tradicional:** Los humanos planifican, escriben el código, y prueban. La IA asiste sugiriendo líneas o autocompletando.
2. **Modelo AI-DLC:** La IA planifica (bajo dirección), escribe el código y genera las pruebas. Los humanos validan los planes, revisan la arquitectura y aprueban la implementación.

Esta inversión requiere una reestructuración de las fases de desarrollo. El AI-DLC condensa las múltiples etapas del SDLC en tres fases de alto impacto: Inception (Incepción), Construction (Construcción) y Operations (Operaciones), cada una gobernada por rituales específicos de colaboración denominados "Mob".

1.2 Fase 1: Inception y el Ritual de "Mob Elaboration"

La fase de Incepción es quizás la más crítica en el AI-DLC, ya que aborda el problema de la ambigüedad en los requisitos. En el desarrollo tradicional, la traducción de una necesidad de negocio (ej. "Necesitamos un sistema de login seguro") a requisitos técnicos es un proceso manual propenso a errores de comunicación.

El Ritual: Mob Elaboration
En el AI-DLC, este proceso se formaliza mediante el ritual de Mob Elaboration (Elaboración en Grupo). A diferencia de una reunión de planificación de sprint tradicional, la "Mob Elaboration" incluye a la IA como un participante activo y central.

- **Actores:** Product Owner (Dueño del Producto), Arquitecto de Software, Desarrolladores Senior y el Agente de IA (Kiro/Amazon Q).
- **Entrada (Input):** Una intención de negocio de alto nivel expresada en lenguaje natural.

- **Proceso:**
  1. La IA analiza la intención y la cruza con el conocimiento existente del sistema (arquitectura actual, deuda técnica, estándares de seguridad).
  2. La IA genera proactivamente una serie de preguntas aclaratorias ("¿El login debe cumplir con SOC2?", "¿Soportaremos autenticación biométrica en la fase 1?").
  3. El equipo humano responde en tiempo real, refinando el contexto.
  4. La IA genera instantáneamente artefactos detallados: Historias de Usuario, Criterios de Aceptación (frecuentemente en notación EARS) y Diagramas de Flujo.
- **Salida (Output):** Un conjunto de requisitos validados y "computables". No se escribe código hasta que estos artefactos sean aprobados explícitamente por el humano.

Este ritual elimina la "página en blanco" y asegura que la IA tenga un contexto completo antes de intentar cualquier implementación, reduciendo drásticamente las alucinaciones derivadas de instrucciones vagas.

1.3 Fase 2: Construction y el Ritual de "Mob Construction"

Una vez que los requisitos están "congelados" y validados, comienza la fase de Construcción. Aquí, el concepto de "Sprint" de dos semanas se reemplaza por el concepto de "Bolt", una unidad de trabajo intensa y corta, medida en horas o días.

El Ritual: Mob Construction
El Mob Construction es la sesión donde la planificación técnica ocurre. Antes de escribir una sola línea de código fuente, la IA propone una solución técnica completa.

- **Propuesta de Arquitectura:** La IA genera un documento de diseño (`design.md`) que detalla qué archivos se crearán, qué APIs se modificarán, cómo fluirán los datos y qué bibliotecas se utilizarán.
- **Validación de Riesgos:** El equipo humano revisa este diseño. Esta es la "puerta de calidad" más importante. Es más fácil corregir un error de diseño en un archivo Markdown que refactorizar 2,000 líneas de código erróneo disperso en 20 archivos.
- **Generación de Código:** Solo tras la aprobación del diseño, la IA procede a la "codificación masiva", implementando la lógica, las pruebas unitarias y las pruebas de integración en paralelo.

1.4 Fase 3: Operations y la Retroalimentación Continua

La fase de Operaciones en el AI-DLC se distingue por la integración de la observabilidad directamente en el ciclo de diseño. Se propone un sub-modelo llamado "Observability-Driven AI-DLC". En este esquema, la IA tiene acceso a las métricas de producción (latencia, tasas de error, uso de memoria). Cuando se inicia un nuevo ciclo de Incepción para una funcionalidad existente, la IA consulta automáticamente estos datos. Si una función tiene un rendimiento deficiente en producción, la IA sugerirá proactivamente optimizaciones durante la fase de diseño de la nueva característica, cerrando el ciclo entre Dev y Ops de una manera que DevOps tradicional aspiraba pero raramente lograba de forma automatizada.

1.5 Comparativa Estructural: SDLC Tradicional vs. AI-DLC

La siguiente tabla resume las diferencias fundamentales entre el enfoque establecido y la nueva propuesta, destacando cómo el AI-DLC altera la asignación de recursos y la gestión de riesgos.

Dimensión	SDLC Tradicional (Agile/Scrum)	AI-Driven Development Lifecycle (AI-DLC)
Motor de Ejecución	Ingenieros humanos	Agentes de inteligencia artificial
Rol humano	Creador, escritor de código	Arquitecto, revisor, validador, “portero”
Unidad de tiempo	Sprint (1–4 semanas)	Bolt (horas–días)
Artefactos de diseño	Manuales, frecuentemente desactualizados	Generados por IA, vivos y vinculantes
Gestión de riesgos	Retrospectiva (post-incidente)	Proactiva (durante el diseño vía Mob Construction)
Principal cuello de botella	Velocidad de escritura y coordinación humana	Capacidad humana de revisión y validación de contexto

Dimensión	SDLC Tradicional (Agile/Scrum)	AI-Driven Development Lifecycle (AI-DLC)
Flujo de calidad	QA al final del ciclo	QA continuo mediante “hooks” y validación de especificaciones

2. Kiro IDE: La Implementación Técnica del Paradigma

Mientras que el AI-DLC proporciona la teoría, Kiro IDE es la herramienta construida por AWS para hacer operativa esta teoría. Lanzado en versión preliminar a mediados de 2025 y evolucionando rápidamente hacia su disponibilidad general, Kiro se posiciona no como un simple editor de código, sino como un "IDE Agéntico". La arquitectura de Kiro se basa en VS Code (Code-OSS), lo que garantiza una barrera de entrada baja para los desarrolladores acostumbrados al ecosistema de Microsoft. Sin embargo, Kiro modifica profundamente el flujo de trabajo mediante la introducción del Desarrollo Basado en Especificaciones (Spec-Driven Development - SDD).

2.1 Spec-Driven Development (SDD): Del Chat al Contrato

La innovación más significativa de Kiro es su rechazo al chat efímero como medio principal de construcción de software. En herramientas competidoras, el contexto se pierde cuando se cierra la ventana de chat o se excede el límite de tokens. En Kiro, el contexto se cristaliza en archivos Markdown persistentes que viven en el repositorio junto con el código. Estos archivos forman una trinidad de artefactos que guían al agente de IA:

1. **requirements.md (El Qué):** Define la intención del producto. Kiro utiliza la notación EARS (Easy Approach to Requirements Syntax) para estructurar estos requisitos.
    - **Sintaxis EARS:** La notación EARS fuerza una estructura lógica (ej. "Mientras el sistema está en estado X, si ocurre el evento Y, entonces el sistema debe realizar Z"). Esto reduce la ambigüedad lingüística que a menudo confunde a los LLMs, asegurando que la IA entienda los límites y condiciones de borde.

2. **design.md (El Cómo):** Este documento es generado por la IA tras analizar los requisitos y el código existente. Actúa como un plano arquitectónico. Detalla los cambios en la base de datos, las nuevas firmas de API, y las dependencias de componentes. Su existencia permite a los humanos auditar la lógica de la solución antes de auditar la sintaxis del código.
3. **tasks.md (El Cuándo):** Una descomposición granular de la implementación en pasos discretos y secuenciales. La IA marca estos pasos a medida que avanza, proporcionando trazabilidad y permitiendo puntos de control donde el humano puede intervenir si la ejecución se desvía.

2.2 Archivos de Dirección (Steering Files): La Constitución del Proyecto

Uno de los desafíos más grandes en el desarrollo asistido por IA es mantener la coherencia estilística y arquitectónica. Un desarrollador humano nuevo tarda meses en aprender las "reglas no escritas" de un proyecto. Una IA genérica no las conoce en absoluto.

Kiro introduce los Steering Files, ubicados en el directorio .kiro/steering/, para resolver esto. Estos archivos actúan como una constitución inmutable que el agente de IA debe "leer" y obedecer antes de cada operación.

Tipología y Jerarquía de Steering Files:
La investigación revela una estructura sofisticada para estos archivos de gobierno:

- **Global Steering:** Reglas que aplican a todos los proyectos del usuario (ej. ~/.kiro/steering/global.md). Pueden incluir preferencias personales como "Siempre usar tipos estrictos en TypeScript" o "Nunca usar any".
- **Workspace Steering:** Reglas específicas del repositorio (`.kiro/steering/`). Son compartidas por todo el equipo a través de Git, asegurando que todos los agentes de IA (independientemente de quién los opere) sigan las mismas normas.

- **Tech-Specific Steering:** El repositorio de "Best Practices" de Kiro sugiere archivos modulares como:
  - `aws-cli-best-practices.md`: Instruye a la IA para usar flags como `--no-cli-pager` para evitar que los scripts se bloqueen esperando input del usuario.
  - `security-best-practices.md`: Reglas duras sobre manejo de secretos y sanitización de inputs.
  - `cdk-best-practices.md`: Patrones arquitectónicos para Infraestructura como Código.

Este mecanismo permite a los líderes técnicos "programar al programador (IA). En lugar de corregir el mismo error en 20 Pull Requests diferentes, el arquitecto añade una regla al Steering File (ej. "Todos los endpoints de la API deben retornar respuestas envueltas en un objeto StandardResponse") y la IA cumple esa regla perpetuamente en el futuro.

2.3 Agent Hooks: Automatización Event-Driven

Más allá de la generación de código, Kiro automatiza el ciclo de vida mediante Agent Hooks (`.kiro/hooks/`). Estos son scripts de configuración que vinculan eventos del IDE con acciones del agente. La capacidad de los hooks transforma el IDE de una herramienta pasiva a un sistema activo de aseguramiento de calidad:

- **Validación en Guardado (on-save):** Al guardar un archivo `.ts`, un hook puede disparar una revisión automática de cumplimiento de estándares, no solo sintácticos (linter), sino semánticos (ej. verificar si la lógica de negocio viola una regla del steering file).
- **Pruebas Automáticas y Auto-reparación:** Un hook configurado como `auto-test-on-save` puede ejecutar las pruebas unitarias afectadas por un cambio reciente. Si la prueba falla, el agente puede intentar corregir el código automáticamente antes de notificar al usuario, creando un ciclo de retroalimentación extremadamente rápido.
- **Sincronización de Documentación:** Un caso de uso crítico es mantener la documentación viva. Un hook puede detectar cambios en las firmas de funciones públicas y actualizar automáticamente el `README.md` o la documentación de la API, eliminando la discrepancia histórica entre código y documentación.

3. Análisis Competitivo: Kiro en el Ecosistema de IDEs Agénticos

La propuesta de Kiro no existe en el vacío. El mercado de "IDEs Agénticos" en 2025 es ferozmente competitivo, con actores consolidados como Cursor y desafiantes innovadores como Windsurf (de Cognition). Entender las diferencias es crucial para evaluar la viabilidad de la propuesta del AI-DLC.

3.1 Kiro vs. Cursor: La Batalla entre Estructura y Velocidad

Cursor se ha establecido como el líder del mercado gracias a su enfoque en la experiencia de usuario (UX) fluida y su modelo de "Composer".

- **Filosofía de Cursor:** Prioriza la velocidad y la baja fricción. Su característica "Composer" permite ediciones rápidas en múltiples archivos a través de un chat flotante.
- **Diferenciador de Kiro:** Kiro sacrifica conscientemente la velocidad inicial en favor de la corrección y la estructura. El proceso de generar Specs y esperar validaciones es más lento que el chat instantáneo de Cursor, pero está diseñado para proyectos donde el costo del error es alto (Enterprise).
- **Modelo de Precios y Transparencia:** Una ventaja competitiva clave identificada en Kiro es su transparencia de costos. Mientras Cursor ha enfrentado críticas por cambios opacos en sus modelos de uso, Kiro ofrece un "pool de créditos unificado" y muestra el costo exacto en créditos de cada prompt antes de ejecutarlo, una característica muy demandada por los gerentes de ingeniería preocupados por el presupuesto.

3.2 Kiro vs. Windsurf: Indexación Implícita vs. Declaración Explícita

Windsurf, impulsado por el agente "Cascade", representa otra filosofía competidora.

- **Enfoque de Windsurf:** Utiliza una indexación profunda del código ("Deep Context") y observa los comandos de la terminal para entender el estado del proyecto. Intenta "adivinar" la intención del usuario basándose en la totalidad de la base de código existente.

- **Enfoque de Kiro:** En lugar de adivinar basándose en el pasado (código existente), Kiro exige que el usuario declare el futuro explícitamente a través de los archivos `design.md` y `requirements.md`.
- **Implicación:** El enfoque de Windsurf es mágico cuando funciona, pero difícil de depurar cuando falla (¿por qué la IA pensó que debía usar esa biblioteca antigua?). El enfoque de Kiro es más determinista: si la IA comete un error, generalmente se puede rastrear a una instrucción ambigua en el Spec o en el Steering File.

3.3 Tabla Comparativa de Funcionalidades

La siguiente tabla sintetiza las capacidades técnicas de las tres plataformas principales según los datos recopilados.

Característica	AWS Kiro IDE	Cursor (Anysphere)	Windsurf (Cognition)
Modelo de interacción	Spec-driven (contratos, EARS)	Chat-driven (Composer, "vibe")	Flow-driven (Cascade, awareness de terminal)
Gestión de contexto	Archivos explícitos (requirements.md)	Indexación de repo + historial de chat	"Deep context" + observación de comandos
Gobernanza	Steering files (`.kiro/steering`)	Reglas (`.cursorrules`)	Memorias y context sharing
Automatización	Agent hooks (event-driven)	Background agents (limitado)	Terminal observation
Audiencia objetivo	Equipos empresariales, arquitectos	Devs individuales, startups	Equipos de alto rendimiento, early adopters
Modelo de IA subyacente	Agnóstico (Bedrock, Claude, etc.)	Agnóstico (Claude, GPT-4, modelos propios)	Familia de modelos SWE-1 (propios)

Característica	AWS Kiro IDE	Cursor (Anysphere)	Windsurf (Cognition)
Transparencia de costos	Alta (medición decimal de créditos)	Media (suscripción plana/uso)	N/D

4. Retos Críticos y Recepción del Mercado

A pesar de la solidez teórica del AI-DLC, la implementación práctica enfrenta desafíos significativos, tanto técnicos como culturales.

4.1 El Fenómeno del "Extraño en el Código" (Stranger in the Code)

Una crítica recurrente en la comunidad de desarrolladores (Reddit, Hacker News) es la alienación que produce el uso de herramientas como Kiro.

- **El Problema:** Cuando un desarrollador escribe código manualmente, construye un mapa mental detallado de la lógica, las dependencias y los flujos de datos. Cuando Kiro genera 2,000 líneas de código en base a un Spec, el desarrollador (incluso si revisó el `design.md`) carece de ese mapa mental granular.
- **La Consecuencia:** En el momento en que surge un bug complejo (especialmente condiciones de carrera o problemas de concurrencia), el desarrollador se siente como un "extraño" en su propio proyecto. La depuración se convierte en una tarea de arqueología de código, mucho más lenta y frustrante que depurar código propio.
- **Mitigación Fallida:** Aunque los rituales de "Mob Construction" intentan mitigar esto forzando revisiones, la realidad psicológica es que la revisión pasiva nunca genera la misma profundidad de comprensión que la creación activa.

4.2 Estudio de Caso: El Fallo de la "Restaurant App"

Un análisis independiente documentó un intento de construir una aplicación de rastreo de restaurantes utilizando Kiro. Los resultados fueron mixtos e ilustrativos de las limitaciones actuales:

- **Éxito en Requisitos:** Kiro sobresalió en la fase de Incepción, deduciendo características implícitas (como filtros de búsqueda y ordenamiento) que el usuario no había solicitado explícitamente pero que eran lógicas para el dominio.
- **Fallo en Ejecución:** A pesar de tener un plan sólido, el código generado no era funcional "out of the box". Kiro tuvo dificultades específicas con la integración de tecnologías AWS (irónicamente), generando código CDK con dependencias circulares y Lambdas en TypeScript mal empaquetadas.
- **Fricción Operativa:** La IA intentaba ejecutar comandos de terminal en directorios incorrectos, requiriendo una supervisión constante ("babysitting") por parte del desarrollador. Esto contradice la promesa de "agentes autónomos" y subraya que, en 2025, la IA todavía requiere un operador experto.

4.3 La Solicitud de VS Code (Issue #277729)

La comunidad ha reaccionado a estas innovaciones solicitando que las características de Kiro se incorporen al estándar de VS Code.

- **La Propuesta:** El usuario ivan1016017 abrió una solicitud de funcionalidad ("feature request") en el repositorio de Microsoft, pidiendo flujos de workflows estructurados inspirados en Kiro (`requirements.md` , `design.md`).
- **Lectura del Mercado:** Esto indica una validación del método (Spec-Driven Development) pero una resistencia a la fragmentación de las herramientas. Los desarrolladores quieren la estructura del AI-DLC, pero prefieren tenerla dentro de su entorno nativo (VS Code / GitHub Copilot) en lugar de migrar a un fork propietario de AWS como Kiro.

5. Análisis Económico y Operativo

La transición al AI-DLC implica no solo cambios técnicos, sino una reestructuración económica de los departamentos de ingeniería.

5.1 De CapEx Humano a OpEx Computacional

El modelo económico del desarrollo de software está cambiando. Tradicionalmente, el costo principal era el salario de los desarrolladores (Capital Humano). Con herramientas como Kiro, una parte de ese costo se desplaza hacia el consumo de créditos de inferencia de IA (Gasto Operativo).

- **Costos Ocultos:** Aunque la IA reduce las horas hombre para generar código, aumenta el consumo de recursos computacionales. Tareas complejas en "Modo Spec" que requieren razonamiento profundo y planificación pueden consumir cantidades significativas de tokens. El modelo de precios de Kiro (Free, Pro, Power) y su sistema de "sobrecarga" ($0.04 por crédito adicional) sugieren que las empresas deben presupuestar la "energía cognitiva" de la IA como un nuevo rubro en sus finanzas.

5.2 Bifurcación de Habilidades Laborales

El AI-DLC acelera la obsolescencia del rol de "desarrollador junior" tradicional, cuya tarea principal era escribir código repetitivo o boilerplate.

- **Nueva Demanda:** La demanda se desplaza hacia perfiles que pueden operar eficazmente la fase de Incepción y Construcción: Ingenieros de Requisitos, Arquitectos de Sistemas y Revisores de Código expertos.
- **La Paradoja de la Experiencia:** Surge una preocupación sistémica: si la IA escribe todo el código "simple", ¿dónde adquirirán los desarrolladores junior la experiencia necesaria para convertirse en los arquitectos senior que deben supervisar a la IA? El AI-DLC asume una competencia experta en la revisión que tradicionalmente solo se adquiría mediante años de escritura de código.

6. Conclusiones y Perspectivas Futuras

La investigación sobre el progreso actual de la propuesta del AI-Driven Development Lifecycle revela una tecnología que ha madurado rápidamente desde un concepto teórico hasta una implementación tangible en Kiro IDE.

Hallazgos Principales

1. **Validación del Problema:** Existe un consenso industrial de que el "Vibe Coding" no es escalable para software empresarial. La necesidad de estructura, persistencia y gobernanza es real y urgente.
2. **Solidez Metodológica:** La estructura de tres fases (Inception, Construction, Operations) y los rituales de "Mob" proporcionan un marco robusto que mitiga los riesgos de alucinación y deriva de requisitos.
3. **Madurez Tecnológica Pendiente:** Aunque la visión es clara, la ejecución actual en Kiro IDE todavía sufre de problemas de fiabilidad ("bugs" en código generado, errores de contexto en terminal), lo que impide una adopción "desatendida". La supervisión humana sigue siendo intensiva.
4. **Innovación en Gobernanza:** Los conceptos de Steering Files y Agent Hooks son quizás las contribuciones más duraderas de esta propuesta. Independientemente del éxito de Kiro como producto, es muy probable que estos mecanismos de "constitución de proyecto" se conviertan en estándares de la industria, adoptados eventualmente por competidores como GitHub y Microsoft.

El Camino Hacia Adelante (2026+)

El futuro del AI-DLC apunta hacia la Validación Neuro-Simbólica. Las próximas iteraciones de estos agentes no solo predecirán el siguiente token estadísticamente probable (Enfoque Neuronal), sino que utilizarán razonamiento lógico y verificación formal (Enfoque Simbólico) para asegurar matemáticamente que el código generado cumple con los requisitos definidos en los Specs.

En conclusión, el AI-DLC propuesto por AWS no es una simple mejora incremental de las herramientas existentes, sino un intento ambicioso de redefinir la ontología misma del desarrollo de software. Si bien enfrenta resistencia cultural y desafíos técnicos de corto plazo, el movimiento hacia un desarrollo guiado por especificaciones y ejecutado por agentes parece inevitable para la ingeniería de software a gran escala.

Si quieres hacer parte de una comunidad que prioriza la ingeniería sobre la magia de vibe coding, mira las clases y únete al grupo de WhatsApp Tribu iA/Vibe Engineering.